- Directivas
- Declaraciones
- Scriptles
- Variables Predefinidas
- Procesamiento de un JSP
- El motor JSP
- Ejemplos de código
-

Directivas - Declaraciones y Scriptles

Directivas

Las directivas JSP son instrucciones procesadas por el motor JSP cuando la página JSP se traduce a un servlet. Las directivas usadas en este ejemplo le dicen al motor JSP que incluya ciertos paquetes y clases. Las directivas están encerradas entre etiquetas de directiva <%@ y

%>.

<%@ page import="javax.naming.*" %>

<%@ page import="javax.rmi.PortableRemoteObject" %>

<%@ page import="Beans.*" %>

Declaraciones

Las declaraciones JSP nos permiten configurar variables para su uso posterior en expresiones o scriptlets. También podemos declarar variables dentro de expresiones o scriptlets en el momento de usarlas. El ámbito es toda la página JSP, no hay concepto de variables de ejemplar. Es decir, no tenemos que declarar variables de ejemplar para usar en más de una expresión o scriptlet. Las declaraciones van encerradas entre etiquetas de declaración <%! y

%>. Podemos tener varias declaraciones. Por ejemplo,

<%! double bonus; String text; %> .

<%! String strMult, socsec; %>

<%! Integer integerMult; %>

<%! int multiplier; %>

<%! double bonus; %>

Scriptlets

Los scriptlets JSP nos permiten embeber segmentos de código java dentro de una página JSP. El código embebido se inserta directamente en el servlet generado que se ejecuta cuando se pide la página. Este scriptlet usa las variables declaradas en las directivas descritas arriba. Los Scriptlets van encerrados entre etiquetas <% y %>.

```
<%

strMult = request.getParameter("MULTIPLIER"); socsec =
request.getParameter("SOCSEC"); integerMult = new Integer(strMult); multiplier =
integerMult.intValue();

bonus = 100.00;

%>
```

Variables Predefinidas

Un scriptlet puede usar las siguientes variables predefinidas: session, request, response, out, e in. Este ejemplo usa la variable predefinida request, que es un objeto HttpServletRequest. De igual forma, response es un objeto HttpServletResponse, out es un objeto PrintWriter, e in es un objeto BufferedReader. Las variables predefinidas se usan en los scriptlets de la misma forma que se usan en los servelts, excepto que no las declaramos.

<%

strMult = request.getParameter("MULTIPLIER"); socsec = request.getParameter("SOCSEC"); integerMult = new Integer(strMult); multiplier = integerMult.intValue();

bonus = 100.00;

%>

Expresiones

La expresiones JSP nos permiten recuperar dinámicamente o calcular valores a insertar directamente en la página JSP. En este ejemplo, una expresión recupera el número de socio desde el bean de entidad Bonus y lo pone en la página JSP.

<H1>Calculo</H1>

Número de socio:

<%= record.getSoc() %>

<P>

calculo: <%= record.getsocio() %>

<P>

Etiquetas específicas de JSP

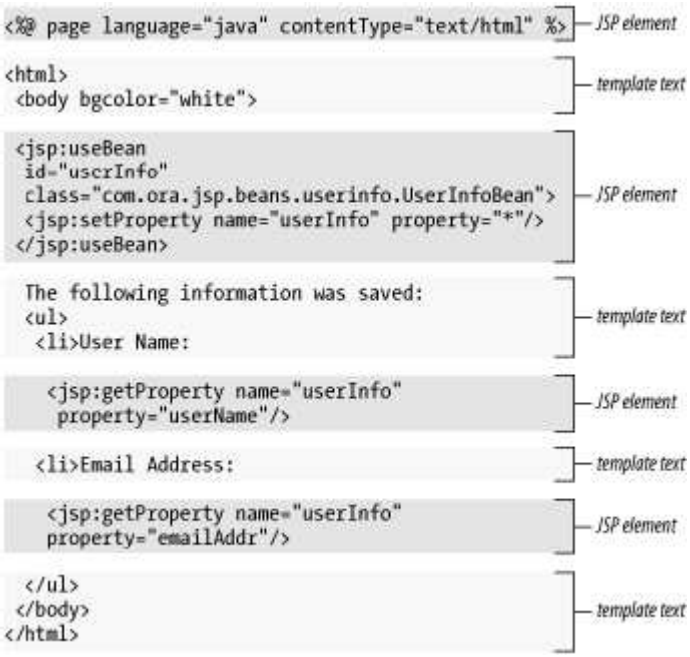
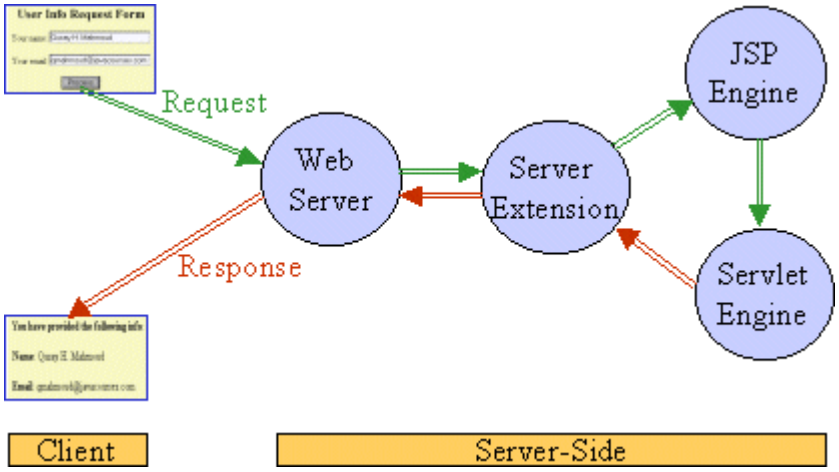


Fig. Estructura de una página JSP

La especificación JavaServer Pages define etiquetas específicas de JSP que nos permiten extender la implementación JSP con nuevas características y ocultar mucha complejidad a los diseñadores visuales que necesitan buscar la página JSP y modificarla

Procesamiento de la página JSP

Cuando se llame a la página (date.jsp), será compilada (por el motor JSP) en un Servlet Java. En este momento el Servlet es manejado por el motor Servlet como cualquier otro Servlet. El motor Servlet carga la clase Servlet (usando un cargador de clases) y lo ejecuta para crear HTML dinámico para enviarlo al navegador, como se ve en la Figura 2. Para este ejemplo, el Servlet crea un objeto Date y lo escribe como un String en el objeto out, que es el stream de salida hacia el navegador.


User Info Request Form
Request
Web Server
Server Extension
JSP Engine
Servlet Engine
Response
You have provided the following info
Client
Server-Side

El Motor de JSP

El motor JSP nos ofrece instancias de un conjunto de clases. Son objetos ya establecidos, que no tenemos más que usar (no hay que instanciarlos). Deben utilizarse dentro del código Java.

Algunos objetos implícitos:

page (*javax.servlet.jsp.HttpJspPage*.): Instancia del servlet de la página. Esto es sólo un sinónimo de this, y no es muy útil en Java. Fue creado como situación para el día que el los lenguajes de script puedan incluir otros lenguajes distintos de Java.

config (*javax.servlet.ServletConfig*): Datos de configuración del servlet.

request (*javax.servlet.http.HttpServletRequest*): Datos de la petición, incluyendo los parámetros. Este es el HttpServletRequest asociado con la petición, y nos permite mirar los parámetros de la petición (mediante getParameter), el tipo de petición (GET, POST, HEAD, etc.), y las cabeceras HTTP entrantes (cookies, Referer, etc.). Estrictamente hablando, se permite que la petición sea una subclase de ServletRequest distinta de HttpServletRequest, si el protocolo de la petición es distinto del HTTP. Esto casi nunca se lleva a la práctica.

response (*javax.servlet.http.HttpServletResponse*): Datos de la respuesta. Este es el HttpServletResponse asociado con la respuesta al cliente. Como el stream de salida tiene un buffer, es legal seleccionar los códigos de estado y cabeceras de respuesta, aunque no está permitido en los servlets normales una vez que la salida ha sido enviada al cliente.

out (*javax.servlet.jsp.JspWriter*): Flujo de salida para el contenido de la página. Este es el PrintWriter usado para enviar la salida al cliente. Sin embargo, para poder hacer útil el objeto response esta es una versión con buffer de PrintWriter llamada JspWriter. Podemos ajustar el tamaño del buffer, o incluso desactivar el buffer, usando el atributo buffer de la directiva page. Se usa casi exclusivamente en scriptlets ya que las expresiones JSP obtienen un lugar en el stream de salida, y por eso raramente se refieren explícitamente a out.

session (*javax.servlet.http.HttpSession*): Datos específicos de la sesión de un usuario. Este es el objeto HttpSession asociado con la petición. Las sesiones se crean automáticamente, por esto esta variable se une incluso si no hubiera una sesión de referencia entrante. La única excepción es usar el atributo session de la directiva page para desactivar las sesiones, en cuyo caso los intentos de referenciar la variable session causarán un error en el momento de traducir la página JSP a un servlet.

application (*javax.servlet.ServletContext*): Datos compartidos por todas las páginas de una aplicación. El ServletContext obtenido mediante getServletConfig().getContext().

pageContext (*javax.servlet.jsp.PageContext*): Datos de contexto para la ejecución de la página. JSP presenta una nueva clase llamada PageContext para encapsular características de uso específicas del servidor como JspWriters de alto rendimiento. La idea es que, si tenemos acceso a ellas a través de esta clase en vez directamente, nuestro código seguirá funcionando en motores servlet/JSP "normales".

exception (*java.lang.Throwable*): Errores o excepciones no capturadas.

Ejemplo:

```
<%
String strParam = request.getParameter("nombre_del_parametro"); out.println( strParam );
%>
```

Algunos ejemplos

Elementos de Script

En el ejemplo date.jsp se usa todo el nombre de la clase Date incluyendo el nombre del paquete, lo que podría llegar a ser tedioso. Si queremos crear un ejemplar de la clase Date usando simplemente: Date today = new Date(); sin tener que especificar el path completo de la clase, usamos la directiva page de esta forma:

Ejemplo #1: fecha.jsp

```
<%@page import="java.util.*" %>
<HTML>
<HEAD>
<TITLE>JSP Example</TITLE>
</HEAD>
<BODY BGCOLOR="ffffcc">
<CENTER>
<H2>Date and Time</H2>
<%
java.util.Date today = new java.util.Date(); out.println("Today's date is: "+today);
%>
</CENTER>
</BODY>
</HTML>
```

Todavía hay otra forma de hacer lo mismo usando la etiqueta <%= escribiendo: Today's date is: <%= new Date() %>

Como podemos ver, se puede conseguir el mismo resultado usando diferentes etiquetas y técnicas. Hay varios elementos de script JSP. Hay algunas reglas convencionales que nos ayudarán a usar más efectivamente los elementos de Script JSP.

- Usamos <% ... %> para manejar declaraciones, expresiones, o cualquier otro tipo de código válido.
- Usamos la directiva page como en <%@page ... %> para definir el lenguaje de script. También puede usarse para especificar sentencias import. Aquí hay un ejemplo:

```
<%@page language="java" import="java.util.*" %>
```

- Usamos <%!%> para declarar variables o métodos. Por ejemplo:

```
<%! int x = 10; double y = 2.0; %>
```

- Usamos <%=%> para definir una expresión y forzar el resultado a un String. Por

ejemplo: <%= a+b %> o <%= new java.util.Date() %>.

- Usamos la directiva include como en <%@ include%> para insertar el contenido

de otro fichero en el fichero JSP principal. Por ejemplo:

```
<%@include file="copyright.html" %>
```

Manejar Formularios

Una de las partes más comunes en aplicaciones de web es un formulario HTML donde el usuario introduce alguna información como su nombre y dirección. Usando JSP, los datos del formulario (la información que el usuario introduce en él) se almacenan en un objeto request que es enviado desde el navegador hasta el contenedor JSP. La petición es procesada y el resultado se envía a través de un objeto response de vuelta al navegador. Estos dos objetos están disponibles implícitamente para nosotros.

Para demostrar como manejar formularios HTML usando JSP, aquí tenemos un formulario de ejemplo con dos campos: uno para el nombre y otro para el email. Como podemos ver, el formulario HTML está definido en un fichero fuente JSP. Se utiliza el método request.getParameter para recuperar los datos desde el formulario en variables creadas usando etiquetas JSP.

La página procesar.jsp imprime un formulario o la información proporcionada por el usuario dependiendo de los valores de los campo del formulario. Si los valores del formulario son null se muestra el formulario, si no es así, se mostrará la información proporcionada por el usuario. Observa que el formulario es creado y manejado por el código del mismo fichero JSP.

Ejemplo #2: procesar.jsp

```
<HTML>
<HEAD>
<TITLE>Formulario Ejemplo</TITLE>
</HEAD>
<BODY BGCOLOR="#ffffcc">
        <% if (request.getParameter("name")==null && request.getParameter("email")== null) { %>
<CENTER>
<H2>User Info Request Form</H2>
<FORM METHOD="GET" ACTION="procesar.jsp">
<P>Nombre: <input type="text" name="name" size=26>
<P>email: <input type="text" name="email" size=26>
<P><input type="submit" value="Process">
</FORM>
</CENTER>
<% } else { %>
<%! String name, email; %>
<% name = request.getParameter("name");
email = request.getParameter("email"); %>
<P><B>Ha introducido la siguiente información</B>:
```

```
<P><B>Nombre</B>: <%= name %>

<P><B>Email</B>: <%= email %>

<% } %>

</BODY>

</HTML>
```

Comentarios: En el contexto de JSP, los JavaBeans contienen la lógica de negocio que devuelve datos a un script en una página JSP, que a su vez formatea los datos devueltos por el componente JavaBean para su visualización en el navegdor. Una página JSP utiliza un componente JavaBean fijando y obteniendo las propiedades que proporciona.

Hay muchos beneficios en la utilización de JavaBeans para mejorar las páginas JSP:

- Componentes Reutilizables: diferentes aplicaciones pueden reutilizar los mismos componentes.
- Separación de la lógica de negocio de la lógica de presentación: podemos modificar la forma de mostrar los datos sin que afecte a la lógica del negocio.
- Protegemos nuestra propiedad intelectual manteniendo secreto nuestro código fuente.

Uso de JavaBeans con JSP

Ahora, veamos como modificar el ejemplo anterior, procesar.jsp para usar JavaBeans. En el formulario anterior había dos campos: name y email. En JavaBeans, son llamados propiedades. Por eso, primero escribimos un componente JavaBean con métodos setX getX, donde X es el nombre de la propiedad. Por ejemplo, si tenemos unos métodos llamados setName y getName entonces tenemos una propiedad llamada name. El ejemplo #3 muestra un componente FormBean.

Los buenos componentes deben poder interoperar con otros componentes de diferentes vendedores. Por lo tanto, para conseguir la reutilización del componente, debemos seguir dos reglas importantes (que son impuestas por la arquitectura JavaBeans):

- Nuestra clase bean debe proporcionar un constructor sin argumentos para que pueda ser creado usando Beans.instantiate.
- Nuestra clase bean debe soportar persistencia implementando el interface

Serializable o Externalizable.

Ejemplo #3: FormBean.java

```
package userinfo; import java.io.*;

    public class FormBean implements Serializable { private String name;
private String email; public FormBean() {
name = null; email = null;
}
  public void setName(String name) { this.name = name;
}
  public String getName() { return name;
}
  public void setEmail(String email) { this.email = email;
}
  public String getEmail() { return email;
}
}
```

Para poder usar el componente FormBean en el fichero JSP, necesitamos ejemplarizar el componente. Esto se hace usando la etiqueta <jsp:useBean>. La siguiente línea

<jsp:setProperty> se ejecuta cuando se ha ejemplarizado el bean, y se usa para inicializar

sus propiedades. En este caso, ambas propiedades (name y email) se configuran usando una sola sentencia. Otra posible forma de configurar las propiedades es hacerlo una a una, pero primero necesitamos recuperar los datos desde el formulario. Aquí tenemos un ejemplo de como configurar la propiedad name:

```
<%! String yourname, youremail; %>

<% yourname = request.getParameter("name"); %>

<jsp:setProperty name="formbean" property="name" value="<%=yourname%>"/>
```

Una vez que se han inicializado las propiedades con los datos recuperados del formulario, se recuperan los valores de las propiedades usando <jsp:getProperty> en la parte else, como se ve en el Ejemplo #4:

Ejemplo #4: procesar2.jsp

```
<jsp:useBean id="formbean" class="userinfo.FormBean"/>

<jsp:setProperty name="formbean" property="*"/>

<HTML>

<HEAD>

<TITLE>Form Example</TITLE>

</HEAD>

<BODY BGCOLOR="#ffffcc">

<% if (request.getParameter("name")==null

&& request.getParameter("email") == null) { %>

<CENTER>

<H2>User Info Request Form </H2>

<form method="GET" action="procesar2.jsp">

<P>
```

```
Your name: <input type="text" name="name" size=27>
<p>
Your email: <input type="text" name="email" size=27>
<P>
<input type="submit" value="Process">
</FORM>
</CENTER>
<% } else { %>
<P>
<B>You have provided the following info</B>:
<P>
<B>Name</B>: <jsp:getProperty name="formbean" property="name"/>
<P>
<B>Email</B>: <jsp:getProperty name="formbean" property="email"/>
<% } %>
</BODY>
</HTML>
```

2. ¿Qué es Spring?

¿Qué es Spring?

Como definición podemos decir que Spring es un framework de código abierto para la creación de aplicaciones empresariales Java, con soporte para Groovy y Kotlin. Tiene una estructura modular y una gran flexibilidad para implementar diferentes tipos de arquitectura según las necesidades de la aplicación.

¿Qué es un framework?

Spring se considera un framework, pero este es un concepto que en ocasiones se confunde, así que vamos a ver qué es un framework.

Una librería es un conjunto de clases, de funciones y de utilidades que nos permiten realizar algunos procesos. Un buen ejemplo de librería sería una librería matemática, a la cual le podemos dar muchos datos y nos puede calcular, por ejemplo, la desviación típica, o le podríamos plantear una integral y la podría resolver.

A diferencia de una librería, un framework es:

- Un conjunto de artefactos software, es decir, que puede incluir una librería, de conceptos y de metodologías.
- Nos proveé de un mecanismo genérico para resolver uno o más problemas de un tipo determinado.
- Es extensible a través de código escrito por los usuarios.
- Ofrece facilidad para el desarrollo y despliegue.

Si tuviéramos que desarrollar una aplicación web, podríamos utilizar un framework que nos facilite la tarea, que nos aporte soluciones a ese desarrollo. Uno de ellos podría ser, por ejemplo Spring MVC, que nos permitiría crear fácilmente una aplicación web, ya que nos aislaría de determinados problemas, como el hecho de crear servlet o registrar las peticiones, así nos podríamos dedicar a lo que realmente importa.

Qué es una aplicación empresarial

Una aplicación empresarial normalmente es:

- Una gran aplicación, es decir, suele ser bastante amplia y orientada a un ámbito comercial o industrial.
- Compleja, es decir, con bastantes funcionalidades, con muchos requisitos y que será utilizada por muchas personas, por lo que debería ser escalable, que pueda crecer

con el tiempo, es decir, que sí hoy tenemos mil usuarios y dentro de un año tenemos un millón, que no haya que rehacer la aplicación de nuevo.
- Distribuida, es decir, que incluso la podamos tener deslocalizada en distintos servidores.
- Crítica, es decir, que no permita o no tolere fallos e incluso, si tuviéramos algún tipo de fallo, que sea capaz de reponerse.
- Orientada a desplegarse dentro de redes corporativas o, el esquema más usual a día de hoy, en internet a través de la nube.
- Centrada en los datos, por lo que todas las funcionalidades se suelen desarrollar en torno a los mismos.
- Intuitiva, de un uso fácil para evitar el rechazo de los usuarios.
- Suele tener, además, unos grandes requisitos de seguridad y de mantenibilidad.

Todas estas serían las características que definirían a una aplicación empresarial.

Spring funciona sobre JVM

Spring funciona con Java, aunque también tiene soporte para otras tecnologías.

Inicialmente se diseñó para trabajar con Java SE y algunas especificaciones o APIs de Java EE, pero a día de hoy trabaja con el JDK 8 y JDK 9. También podemos trabajar con Groovy y tiene soporte para Kotlin.

Spring tiene estructura modular

Spring ya no es solo un framework para la inyección de dependencias, sino que tiene toda una familia de proyectos que abarcan muchos ámbitos: el ámbito de desarrollo de aplicaciones web, aplicaciones web reactivas, seguridad, servicios web, microservicios, Android, etcétera.

Además, dentro de alguno de esos proyectos, podemos encontrar que tiene una estructura modular, es decir, que está orientada a poder tener distintos módulos que agrupan diversas funcionalidades, desde el contenedor de inversión de control, la programación orientada a aspectos, el acceso a datos, etcétera.

Spring es flexible

Spring nos permitir desarrollar todo tipo de aplicaciones diferentes:

Aplicaciones de escritorio, aplicaciones de línea de comando, aplicaciones web clásicas, web reactivas, microservicios…

Aplicaciones que acceden a base de datos vía SQL directamente, a través de algún tipo de ORM, bases de datos NoSQL…

Aplicaciones con esquemas de seguridad clásica donde almacenamos en nuestra base de datos los elementos de seguridad, clave o credenciales, pero también con otros sistemas.

Aplicaciones pequeñas, medianas y grandes, aplicaciones que tienden a ser escalables, aplicaciones que vayamos a gestionar a través de contenedores, aplicaciones que necesitemos desplegar en la nube.

Crear Nuestro Primer proyecto en Spring

Link video Crear proyecto con java Spring: https://youtu.be/qdPEZjhzjlU